

Making a Better List-set

EECS 395 “Rust”

Feb. 18, 2016

Linearizability, formally

History H is *linearizable* if it can be extended to complete history G by

- appending responses to some pending invocations, and/or
- discarding the remaining pending invocations

such that there exists an equivalent legal sequential history S where $\rightarrow_G \subseteq \rightarrow_S$.

Example

$H =$

- A q.enq(3)
- B q.enq(4)
- B q:void
- B q.deq()
- B q:4
- B q.enq(6)

Example

$H =$

- A q.enq(3)
- B q.enq(4)
- B q:void
- B q.deq()
- B q:4
- B q.enq(6)

$G =$

- A q.enq(3)
- B q.enq(4)
- B q:void
- B q.deq()
- B q:4
- A q:void

Example

$H =$

- A q.enq(3)
- B q.enq(4)
- B q:void
- B q.deq()
- B q:4
- B q.enq(6)

$G =$

- A q.enq(3)
- B q.enq(4)
- B q:void
- B q.deq()
- B q:4
- A q:void

$S =$

- B q.enq(4)
- B q:void
- A q.enq(3)
- A q:void
- B q.deq()
- B q:4

S is legal and $\sim G$

Can we do better?

Coarse-grained synchronization:

- Lock the whole object for each operation

Can we do better?

Coarse-grained synchronization:

- Lock the whole object for each operation
- Easy to reason about :-)

Can we do better?

Coarse-grained synchronization:

- Lock the whole object for each operation
- Easy to reason about :-)
- But sequential bottleneck :-)

Four strategies

1. Fine-grained synchronization

Four strategies

1. Fine-grained synchronization

:-) Can synchronize on different parts of object concurrently

Four strategies

1. Fine-grained synchronization

- :-) Can synchronize on different parts of object concurrently
- :-) But lots of locking/unlocking overhead

Four strategies

1. Fine-grained synchronization

- :-) Can synchronize on different parts of object concurrently
- :-(But lots of locking/unlocking overhead

2. Optimistic synchronization

- :-) No need to lock while traversing
- :-(But need to validate, and may require expensive retries

Four strategies

1. Fine-grained synchronization

-) Can synchronize on different parts of object concurrently
- (But lots of locking/unlocking overhead

2. Optimistic synchronization

-) No need to lock while traversing
- (But need to validate, and may require expensive retries

3. Lazy synchronization

-) Less work needed than optimistic synchronization
- (But contended operations still need to retry

Four strategies

1. Fine-grained synchronization
 - :-) Can synchronize on different parts of object concurrently
 - :-(But lots of locking/unlocking overhead
2. Optimistic synchronization
 - :-) No need to lock while traversing
 - :-(But need to validate, and may require expensive retries
3. Lazy synchronization
 - :-) Less work needed than optimistic synchronization
 - :-(But contended operations still need to retry
4. Lock-free synchronization
 - :-) No longer at the mercy of the scheduler
 - :-(But complex, and maybe high overhead

© ⓘ ⓘ This work is licensed under a Creative Commons “Attribution-ShareAlike 3.0 Unported” license.

These slides are derived from the companion slides for *The Art of Multiprocessor Programming*, by Maurice Herlihy and Nir Shavit. Its original license reads:

This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free:**
 - ▶ *to Share* — to copy, distribute and transmit the work
 - ▶ *to Remix* — to adapt the work
- **Under the following conditions:**
 - ▶ **Attribution.** *You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors of that work or this endorse you or your use of the work).*
 - ▶ **Share Alike.** *If you alter, transform, or build upon this work, you may distribute the resulting work onnly under the same, similar or a compatible licese.*
- *For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to*
 - ▶ <http://creativecommons.org/licenses/by-sa/3.0/>.
- *Any of the above conditions can be waived if you get permission from the copyright holder.*
- *Nothing in this license impairs or restricts the author’s moral rights.*